

The Old and the New: Can Physics-Informed Deep-Learning Replace Traditional Linear Solvers?

STEFANO MARKIDIS, KTH Royal Institute of Technology, Sweden

Physics-Informed Neural Networks (PINN) are neural networks encoding the problem governing equations, such as Partial Differential Equations (PDE), as a part of the neural network. PINNs have emerged as a new essential tool to solve various challenging problems, including computing linear systems arising from PDEs, a task for which several traditional methods exist. In this work, we focus first on evaluating the potential of PINNs as linear solvers in the case of the Poisson equation, an omnipresent equation in scientific computing. We characterize PINN linear solvers in terms of accuracy and performance under different network configurations (depth, activation functions, input data set distribution). We highlight the critical role of transfer learning. Our results show that low-frequency components of the solution converge quickly as an effect of the F-principle. In contrast, an accurate solution of the high frequencies requires an exceedingly long time. To address this limitation, we propose integrating PINNs into traditional linear solvers. We show that this integration leads to the development of new solvers whose performance is on par with other high-performance solvers, such as PETSc conjugate gradient linear solvers, in terms of performance and accuracy. Overall, while the accuracy and computational performance are still a limiting factor for the direct use of PINN linear solvers, hybrid strategies combining old traditional linear solver approaches with new emerging deep-learning techniques are among the most promising methods for developing a new class of linear solvers.

1 INTRODUCTION

Deep Learning (DL) has revolutionized the way of performing classification, pattern recognition, and regression tasks in various application areas, such as image and speech recognition, recommendation systems, natural language processing, drug discovery, medical imaging, bioinformatics, and fraud detection, among few examples [8]. However, scientific applications solving linear and non-linear equations with demanding accuracy and computational performance requirements have not been the DL focus. Only until recently, a new class of DL networks, called *Physics-Informed Neural Networks* (PINN), emerged as a very promising DL method to solve scientific computing problems [33–35]. In fact, PINNs are specifically designed to integrate scientific computing equations, such as Ordinary Differential Equations (ODE), Partial Differential Equations (PDE), non-linear and integral-differential equations [28], into the DL network training. In this work, we focus on PINN application to solve a traditional scientific computing problem: the solution of a linear system arising from the discretization of a PDE. We solve the linear system arising from the Poisson equation, one of the most common PDEs whose solution still requires a non-negligible time with traditional approaches. We evaluate the level of maturity in terms of accuracy and performance of PINN linear solver, either as a replacement of other traditional scientific approaches or to be deployed in combination with conventional scientific methods, such as the multigrid and Gauss-Seidel methods [32].

PINNs are deep-learning networks that, after training (solving an optimization problem to minimize a residual function), output an approximated solution of differential equation/equations, given an input point in the integration domain (called collocation point). Before PINNs, previous efforts, have explored solving PDEs with constrained neural networks [18, 31]. The major innovation with PINN is the introduction of a *residual* network that encodes the governing physics equations, takes the output of a deep-learning network (called *surrogate*), and calculates a residual value (a loss function in DL terminology). The inclusion of a *residual* network, somehow, bears a resemblance of those iterative Krylov linear solvers in scientific applications. The fundamental difference is that PINNs calculate differential operators on graphs using automatic differentiation [3] while traditional scientific approaches are based on numerical schemes

for differentiation. As noted in previous works [23, 35], automatic differentiation is the main strength of PINNs because operators on the residual network can be elegantly and efficiently formulated with automatic differentiation. An important point is that the PINN’s *residual* network should not be confused with the popular network architectures, called also *Residual* networks, or *ResNet* in short, where the name derives from using skip-connection or residual connections [8] instead of calculating a residual like in PINNs.

The basic formulation of the PINN training does not require labeled data, e.g., results from other simulations or experimental data, and is unsupervised: PINNs only require the evaluation of the residual function [23]. Providing simulation or experimental data for training the network in a supervised manner is also possible and necessary for so data-assimilation [36], inverse problems [24], super resolution [5, 44], and discrete PINNs [35]. The supervised approach is often used for solving ill-defined problems when for instance we lack boundary conditions or an Equation of State (EoS) to close a system of equations (for instance, EoS for the fluid equations [48]). In this study, we only focus on the basic PINNs as we are interested in solving PDEs without relying on other simulations to assist the DL network training. A common case in scientific applications is that we solve the same PDE with different source terms at each time step. For instance, in addition to other computational kernels, Molecular Dynamics (MD) code and semi-implicit fluid and plasma codes, such as GROMACS [42], Nek5000 [30], and iPIC3D [22], calculate the Poisson equation for the electrostatic and pressure solver [27] and divergence cleaning operations at each cycle.

Once a PINN is trained, the inference from the trained PINN can be used to replace traditional numerical solvers in scientific computing. In this so-called *inference* or *prediction* step, the input includes independent variables like simulation time step and simulation domain positions. The output is the solution of the governing equations at the time and position specified by the input. Therefore, PINNs are a *gridless* method because any point in the domain can be taken as input without requiring the definition of a mesh. Moreover, the trained PINN network can be used for predicting the values on simulation grids of different resolutions without the need of being retrained. For this reason, the computational cost does not scale with the number of grid points like many traditional computational methods. PINNs borrow concepts from popular methods in traditional scientific computing, including Newton-Krylov solvers [14], finite element methods (FEM) [38], and Monte Carlo techniques [39]. Like the Newton-Krylov solvers, PINNs training is driven by the objective of minimizing the residual function and employs Newton methods during the optimization process. Similarly to the FEM, PINN uses interpolation basis (non-linear) functions, called *activation functions* [37] in the neural network fields. Like Monte Carlo and quasi-Monte Carlo methods, PINNs integrate the governing equations using a random or a low-discrepancy sequence, such as the Sobol sequence [40], for the collocation points used during the evaluation the residual function.

The motivation of this work is twofold. First, we evaluate the potential of deploying PINNs for solving linear systems, such as the one arising from the Poisson equation. We focus on solving the Poisson equation, a generalization of the Laplace equation, and an omnipresent equation in scientific computing. Traditionally, Poisson solvers are based on linear solvers, such as the Conjugate Gradient (CG) or Fast Fourier Transform (FFT). These approaches may require a large number of iterations before convergence and are computationally expensive as the fastest methods scale as $O(N_g \log N_g)$, where N_g is the number of grid points in the simulation domain. The second goal of this work is to propose a new class of linear solvers combining new emerging DL approaches with old traditional linear solvers, such as multigrid and iterative solvers.

In this work, we show that the accuracy and the convergence of PINN solvers can be tuned by setting up an appropriate configuration of depth, layer size, activation functions and by leveraging transfer learning. We find that fully-connected surrogate/approximator networks with more than three layers produce similar performance results in

the first thousand training epochs. The choice of activation function is critical for PINN performance: depending on the *smoothness* of the source term, different activation functions provide considerably different accuracy and convergence. Transfer learning in PINNs allow us to initialize the network with the results of another training solving the same PDE with a different source term [46]. The usage of transfer learning considerably speed-up the training of the network. In terms of accuracy and computational performance, a naive replacement of traditional numerical approaches with the direct usage of PINNs is still not competitive with traditional solvers and codes, such as CG implementations in HPC packages [2].

To address the limitations of the direct usage of PINN, we combine PINN linear solvers with traditional approaches such as the multigrid and Gauss-Seidel methods [32, 41]. The DL linear solver is used to solve the linear system on a coarse grid and the solution refined on finer grids using the multigrid V-cycle and Gauss-Seidel solver iterations. This approach allows us to use the DL networking of converging quickly on low-frequency components of the problem solution and rely on Gauss-Seidel to solve accurately high-frequency components of the solution. We show that the integration of DL techniques in traditional linear solvers leads to solvers that are on-par of high-performance solvers, such as PETSc conjugate gradient linear solvers, both in terms of performance and accuracy.

The paper is organized as follows. We first introduce the governing equations, the background information about PINN architecture and showcase the usage of PINN to solve the 2D Poisson equation. Section 3 presents a characterization of PINN linear solver performance when varying the network size, activation functions, and data set distribution and we highlight the critical importance of leveraging transfer learning. We present the design of a Poisson solver combining new emerging DL techniques into the V-cycle of the multigrid method and analyze its error and computational performance in Section 5. Finally, we summarize this study and outline challenges and next step for the future work in Section 6.

2 THE NEW: PHYSICS-INFORMED LINEAR SOLVERS

The PINNs goal is to approximate the solution of a system of one or more differential, possibly non-linear equations, by encoding explicitly the differential equation formulation in the neural network. Without loss of generality, PINN solves the non-linear equation:

$$u(x)_t = \mathcal{N}u(x) = 0, x \in \Omega, t \in [0, T], \quad (1)$$

where u is the solution of the system, u_t is its derivative with respect to time t in the period $[0, T]$, \mathcal{N} is a non-linear differential operator, x is an independent, possibly multi-dimensional variable, defined over the domain Ω . As a main reference equation to solve, we consider the Poisson equation in a unit square domain and Dirichlet boundary conditions throughout this paper:

$$\nabla^2 u(x, y) = f(x, y), (x, y) \in [0, 1] \times [0, 1]. \quad (2)$$

While this problem is linear in nature and PINNs can handle non-linear problems, we focus on the Poisson equation because it is one of the most solved PDEs in scientific applications. The Poisson equation, an example of elliptic PDE, arises in several different fields from electrostatic problems in plasma and MD codes, to potential flow and pressure solvers in Computational Fluid Dynamics (CFD), to structural mechanics problems. Elliptic problems are one of the Achilles' heels for scientific applications [25]. While relatively fast and straightforward - albeit subject to numerical constraints - computational methods exist for solving hyperbolic and parabolic problems, e.g. explicit differentiation, traditionally the solution of elliptic problems requires linear solvers, such as Krylov (CG or GMREs) solvers or FFT. Typically, in scientific applications, the simulation progresses through several time steps, where a Poisson equation

with same boundary conditions and different source term $f(x, y)$ (typically not considerably different from the source term of the previous time step) is solved.

In its basic formulation, PINNs combine two networks together: an *approximator* or *surrogate* network and a residual network (see Figure 1) [35]. The approximator/surrogate network undergoes training and after it provides a solution \tilde{u} at a given input point (x, y) , called *collocation point*, in the simulation domain. The residual network encodes the governing equations and it is the distinctive feature of PINNs. The residual network is not trained and its only function is to provide the approximator/surrogate network with the residual (*loss function* in DL terminology):

$$r = \nabla^2 \tilde{u}(x, y) - f(x, y). \quad (3)$$

Differently from traditional methods often relying on finite difference approximation, the derivatives on the residual network graph, e.g., $\nabla^2 \tilde{u}(x, y)$ in Equation 3, are calculated using the so-called *automatic differentiation*, or autodiff, that leverages the chain rule [3] applied to the operations defined on the network nodes. In the solution of the Poisson Equation, the Laplacian operator is expressed as two successive first-order derivatives of \tilde{u} in the x and y directions and their summation (see the blue network nodes in Figure 1).

In the inference/prediction phase, only the surrogate network is used to calculate the solution to the problem (remember that the residual network is only used in the training process to calculate the residual).

The approximator/surrogate network is a feedforward neural network [8]: it processes an input x via l layer of units (called also *neurons*). The approximator/surrogate network expresses affine-linear maps (Z) between units and scalar non-linear activation functions (a) within the units:

$$\tilde{u}(x) = Z_l \circ a \circ Z_{l-1} \circ a \dots \circ a \circ Z_2 \circ a \circ Z_1(x). \quad (4)$$

In DL, the most used activation functions are Rectified Linear Unit (ReLU), tanh, swish, sine, and sigmoid functions. See Ref. [37] for an overview of the different activation functions. As shown by Ref. [23], PINNs requires sufficiently smooth activation functions. **PINNs with ReLU and other non-smooth activation functions, such as ELU and SELU (Exponential and Scaled Exponential Linear Units) are not “consistent/convergent” methods:** in the limit of an infinite training dataset a well-trained PINN with ReLU-like activation functions, the solution does not converge to the exact solution [23]. This theoretical result is also confirmed by our experiments using ReLU-like activation functions. For this reason, we do not use ReLU-like activation functions in PINNs.

The affine maps Z are characterized by the weights and biases of the approximator/surrogate network:

$$Z_l x_l = W_l x_l + b_l, \quad (5)$$

where W_l is a *weight* matrix for the layer l and b is the *bias* vector. In PINNs, the weight values are initialized using the *Xavier* (also called *Glorot* when using the last name of the inventor instead) procedure [17].

Typically, the PINN approximator/surrogate networks are fully connected networks consisting of 4-6 hidden layers(H) and 50-100 units per layer, similarly to the network in Figure 1. There are also successful experiments using convolutional and recurrent layers [7, 26] but the vast majority of existing PINNs rely on fully-connected layers. In this work, we focus on studying the performance of fully-connected PINN.

The residual network is responsible for encoding the equation to solve and provide the loss function to the approximator network for the optimization process. In PINNs, we minimize the Mean Squared Error (MSE) of the residual

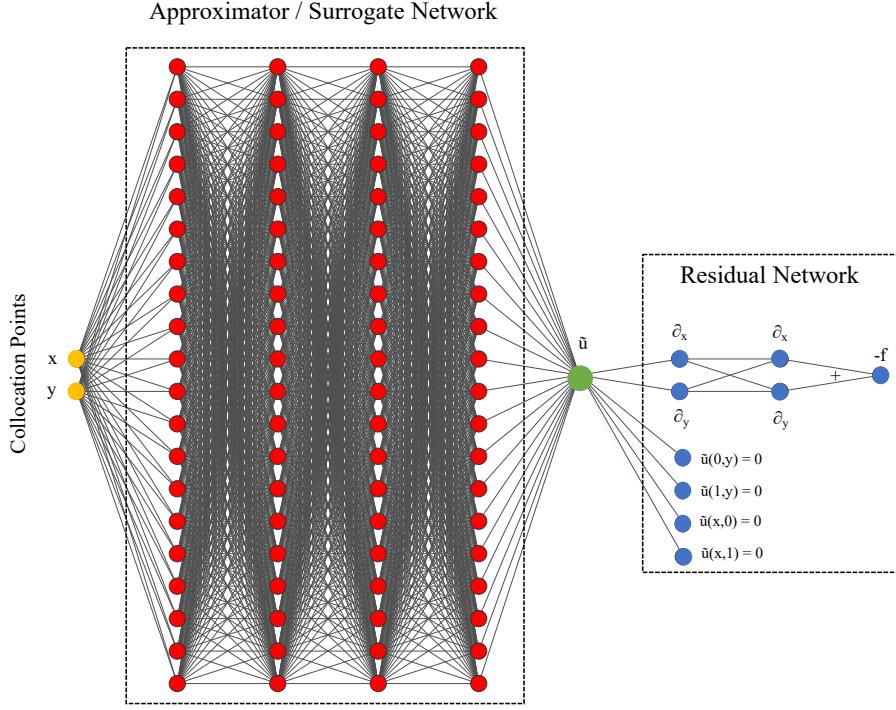


Fig. 1. A PINN to solve a Poisson problem $\partial_x^2 u(x, y) + \partial_y^2 u(x, y) = f(x, y)$ with associated Dirichlet boundary conditions. PINN consists of two basic interconnected networks. The first network (red vertices) provides a surrogate or approximation of the problem solution u . The network takes as input a point in the problem domain (x, y) and provides an approximate solution \tilde{u} . This network weights and biases are trainable. The second network (blue vertices) takes the approximate solution from the first network and calculates the residual that is used as loss function to train the first network. The residual network includes the governing equations, boundary conditions and initial conditions (not included in the plot as the Poisson problem does not require initial conditions).

(Equation 3):

$$MSE_r = \frac{1}{N_{x_i, y_i}} \sum |r(x_i, y_i)|^2, \quad (6)$$

where N_{x_i, y_i} is the number of collocation points. In PINNs, the collocation points constitute the training dataset. Note that MSE_r depends on the size of the training of the dataset (N_{x_i, y_i}), e.g., the number of collocation points. In practice, a larger number of collocation points leads to an increased MSE value. MSE_r depends also on on the distribution of our collocation points. The three most used dataset distributions are: uniform (the dataset is uniformly spaced on the simulation domain as on a uniform grid), pseudo-random (collocations points are sampled using pseudo-random number generator) and Sobol (collocation points are from the Sobol low-discrepancy sequence). Typically, the default training distribution for PINNs is Sobol, like in quasi-Montecarlo methods.

Recently, several PINN architectures have been proposed. PINNs differentiate on how the residual network is defined. For instance, fPINN (fractional PINN) is a PINN with a residual network capable of calculating residuals of governing equations including fractional calculus operators [28]. fPINN combines automatic differentiation with numerical discretization for the fractional operators in the residual network. fPINN extends PINN to solve integral and differential-integral equations. Another important PINN is vPINN (variational PINN): they include a residual network

that uses the variational form of the problem into the loss function [15] and an additional shallow network using trial functions and polynomials and trigonometric functions as test functions. A major advantage with respect to basic PINNs is that in the analytical calculation by integrating by parts the integrand in the variational form, we can the order of the differential operators represented by the neural networks, speeding up the training and increasing PINN accuracy. hp-VPINN is an extension of vPINN that allows hp-refinement via domain decomposition as h-refinement and projection onto space of high order polynomials as p-refinement [16]. In this work, we use the original residual network as shown in Figure 1.

In the training phase, an optimization process targeting the residual minimization determines the weights and biases of the surrogate network. Typically, we use two optimizers in succession: the Adam optimizer as first and then a Broyden-Fletcher-Goldfarb-Shanno (BFGS) optimizer [6]. BFGS uses the Hessian matrix (curvature in highly dimensional space) to calculate the optimization direction and provides more accurate results. However, if used directly without using the Adam optimizer can rapidly converge to a local minimum (for the residual) without exiting. For this reason, the Adam optimizer is used first to avoid local minima, and then the solution is refined by BFGS. We note that the typical BFGS used in PINNs is the L-BFGS-B: L-BFGS is a limited-memory version of BFGS to handle problems with many variables, such as DL problems; the BFGS-B is a variant of BFGS for bound constrained optimization problems. In our work, we tested several optimizers, including Newton and Powell methods, and found that L-BFGS-B provides by far the highest accuracy and faster convergence in all our test problems. **L-BFGS-B is currently the most critical technology for PINNs.**

An *epoch* comprises all the optimizer iterations to cover all the datasets. In PINNs, typically, thousands of epochs are required to achieve accurate results. By nature, PINNs are under-fitted: the network is not complex enough to accurately capture relationships between the collocation points and solution. Therefore, an extensive dataset increase improves the PINN performance; however, the computational cost increases raising the data set size.

One crucial point related to PINNs is whether a neural network can approximate simultaneously and uniformly the solution function and its partial derivatives. Ref. [19] shows that feed-forward neural nets with enough neurons can achieve this task. A formal analysis of the errors in PINNs is presented in Refs. [19, 23].

An important fact determining the convergence behavior of the DL networks and PINN linear solvers is the Frequency-principle (F-principle): DNNs often fit target functions from low to high frequencies during the training process [47]. The F-principle implies that in PINNs, the low frequency / large scale features of the solution emerge first, while it will take several training epochs to recover high frequency / small-scale features. This

Despite the recent introduction of PINNs, several PINN frameworks for PDE solutions exist. All the major PINN frameworks are written in Python and rely either on TensorFlow [1] or PyTorch [29] to express the neural network architecture and exploit auto-differentiation used in the residual network. Together with TensorFlow, SciPy [43] is often used to use high-order optimizers such as L-BFGS-B. Two valuable PINN Domain-Specific Languages (DSL) are DeepXDE [19] and sciANN [11]. DeepXDE is an highly customizable framework with TensorFlow 1 and 2 backend and it supports basic and fractional PINNs in complex geometries. sciANN is a DSL based on and similar to Keras [10]. In this work, we use the DeepXDE DSL.

2.1 An Example: Solving the 2D Poisson Equation with PINN

To showcase how PINNs work and provide a baseline performance in terms of accuracy and computational cost, we solve a Poisson problem in the unit square domain with a source term $f(x, y)$ that is smooth, e.g., differentiable, and

contains four increasing frequencies:

$$f(x, y) = \frac{1}{4} \sum_{k=1}^4 (-1)^{k+1} 2k \sin(k\pi x) \sin(k\pi y). \quad (7)$$

We choose such a source term as it has a simple solution and to show the F-principle's impact on the convergence of PINN to the numerical solution: we expect the lower frequency components, e.g., $k = 1$, to converge faster than the higher frequency components present in the solution ($k = 2, 3, 4$).

We use a fully-connected four-layer PINN with a \tanh activation function for the approximator/surrogate network for demonstration purposes and without a loss of generality. The input layer consists of two neurons (the x and y coordinates of one collocation point), while each hidden and output layers comprise 50 neurons and one neuron, respectively. The weights of the network are initialized with the Xavier method. As a reminder, the approximator/surrogate network's output is the approximate solution to our problem. The residual network is a graph encoding the Poisson equation and source term and provides the loss function (Equation 6) to drive the approximator/surrogate network's optimization. At each, a collocation point within the problem domain is drawn from the Sobol sequence. The training data set consists of 128×128 collocation points on the domain and additional 4,000 collocation points on the boundary for a total of 20,384 points. We train the approximator/surrogate network 10,000 of Adam optimizer epochs with a learning rate λ equal to 0.001 (the magnitude of the optimizer vector along the direction to minimize the residual), followed by 13,000 epochs of L-BFGS-B optimizer. We use the DeepXDE DSL for our PINN implementation.

Figure 2 shows the Poisson equation's approximate solution with the source term of Equation 7 at different epochs, the training error, and the error of the PINN solution after the training is completed. The Figure 2 top panels present the contour plot of the approximator/surrogate solution on a 128×128 uniform grid after 500, 5,000 and 23,000 epochs. To determine the solution at each epoch, we take the approximator/surrogate network and perform inference/prediction using the points of the 128×128 uniform grid. By analyzing the approximate solutions' evolution (top panels of Figure 2), it is clear that the PINN resolves the low-frequency component present in the solution: a yellow band appears along the diagonal of the plot while local peaks (small islands in the contour plot) are not resolved. As the training progresses, localized peaks associated with the source term's high-frequencies appear and are resolved. The bottom right panel of Figure 2 shows a contour plot of the error after the training is completed. The maximum pointwise error is approximately 5E-3. We note that a large part of the error is located in the proximity of the boundaries. This issue results from the *vanishing-gradient* problem [45]: unbalanced gradients back-propagate during the model training. This issue is similar to the numerical *stiffness* problem when using traditional numerical approaches. One of the effective technique to mitigate the *vanishing-gradient* problem is to employ locally (to the layers or the node) adaptive activation functions [12]. Additional techniques for mitigating *vanishing-gradient* problem are the usage of ReLU activations functions and batch normalization.

The bottom panel of Figure 2 shows the training error's evolution calculated with Equation 6. In this case, the initial error is approximately 1.08E2 and decreases up to 2.79E-5 at the end of the training. The initial error mainly depends on the training data set size: small input data sets reduce training error that does not translate to higher accuracy in the solution of the problem. However, the training is a reasonable metric when comparing the PINN performance when using the same data set size.

By analyzing the evolution of the training error, it is clear that the Adam optimizer training error stabilizes approximately in the range of 5E-3 - 1E-2 after 2,000 epochs, and we do not observe any evident improvement after 2,000 epochs of Adam optimization. The L-BFGS-B optimizer leads the error from 5E-3 - 1E-2 to 2.79E-5 and is responsible for

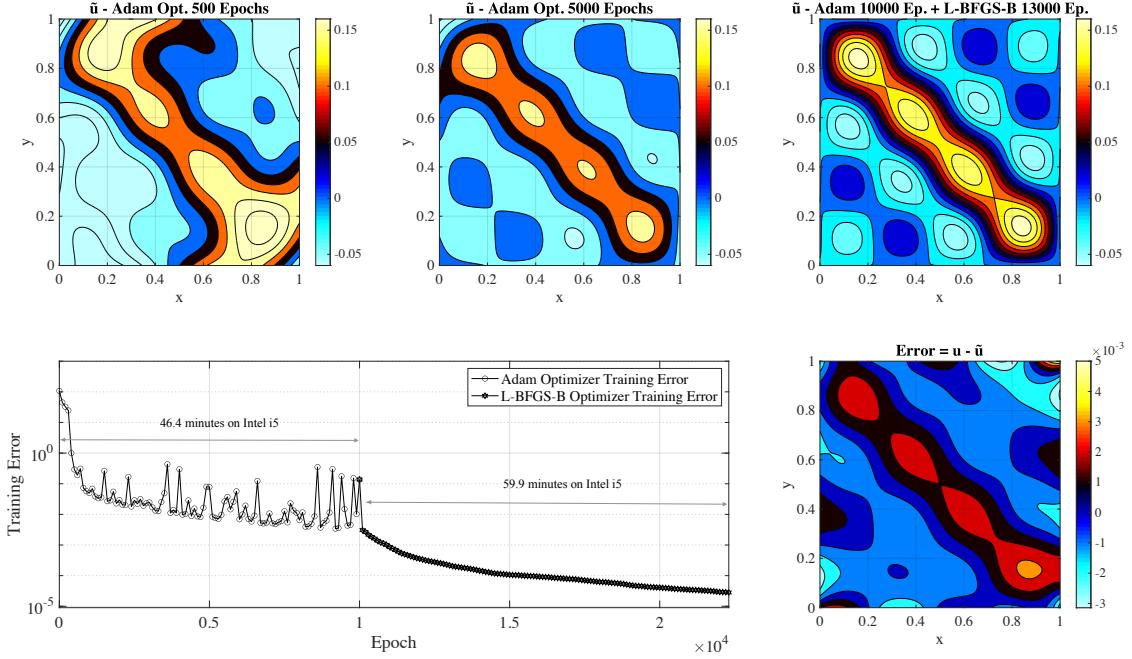


Fig. 2. The top panels show the solution of the Poisson equation at different epochs using a PINN. The bottom panel shows the training error for an initial training with Adam’s optimizer (10,000 epochs), followed by L-BFGS-B (13,000 epochs). The plot also includes the total time for training the PINN on a dual-core Intel i5 processor. The right bottom subplot presents the error of the final solution compared to the exact solution.

the major decrease of the training error. However, we remind that L-BFGS-B is not used at the beginning of the training as it can converge quickly to a wrong solution (a local minimum in the optimization problem).

To provide an idea of the PINN training’s overall computation cost, we also report the total time for training the PINN in this basic non-optimized configuration on a dual-core Intel i5 2.9 GHz CPU. The total training execution time is 6,380 seconds, corresponding to approximately 1.5 hours. For comparison, the solution of the same problem with a uniform grid size 128×128 on the same system with the `petsc4py` CG solver [2, 4] requires 92.28 seconds to converge to double-precision machine epsilon. Basic PINN’s direct usage to solve the Poisson problem is limited for scientific application given the computational cost and the relatively low accuracy. In the next sections, we investigate which factors impact the PINN performance and its accuracy. We design a PINN-based solver to have comparable performance to state-of-the-art linear solvers such as `petsc4py`.

3 CHARACTERIZING PINNS AS LINEAR SOLVERS

To characterize the PINNs performance for solving the Poisson equation, we perform several parametric studies varying the approximator/surrogate network size, activation functions, and training data size and distribution. We also investigate the performance enhancement achieved by using the transfer learning technique to initialize with the network weights obtained solving the Poisson equation with a different source term [46]. During our experiments, we found that two relatively different configurations of the network are required in the case of the source term of the

Poisson equation is smooth on non smooth, e.g. non-differentiable. For this reason, we choose two main use cases to showcase the impact of different parameters. For the smooth source term case, we take the source term from Equation 7 (the example we showcased in the previous section). For the non-smooth source term case, we take a source term that is zero everywhere except for the points enclosed in the circle, centered in $(0.5, 0.5)$ with radius 0.2:

$$f(x, y) = 1 \text{ for } \sqrt{(x - 0.5)^2 + (y - 0.5)^2} \leq 0.2. \quad (8)$$

As baseline configuration, we adopt the same configuration described in the previous section: a fully-connected network with four hidden layers of 50 units, and \tanh activation function. The data set consists of 128×128 collocation points in the domain and 4,000 points on the boundary. Differently from the previous configuration, we reduce the training epochs to 2,000 for the Adam Optimizer (the training error do not decrease after 2,000 epochs) and 5,000 for the L-BFGS-B optimizer.

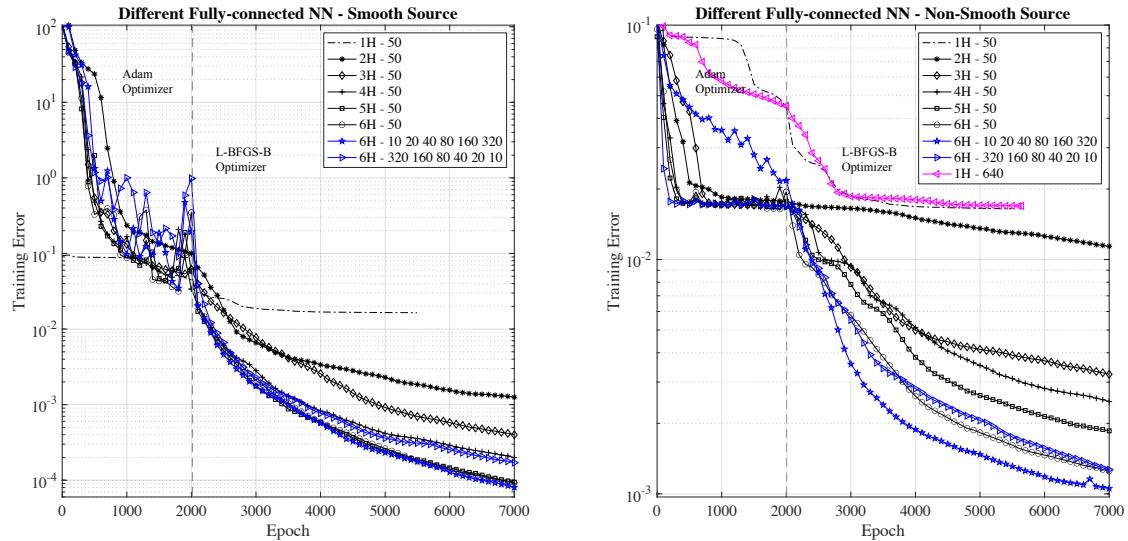


Fig. 3. Training error for different fully-connected PINN depth: one (1H), two (2H), three (3H), four (4H), five (5H) and six (6H) hidden layers with 50 neurons each. We also consider the training error for PINNs with six hidden layers and 10-20-40-80-160-320 and 320-160-80-40-20-10 units per hidden layer, respectively.

The first experiments we perform is to evaluate the impact of the network size (depth and units per layer) on the training error. To understand the impact of surrogate neural network depth, we perform training with layers of 50 neurons with one (1H), two (2H), three (3H), four (4H), five (5H) and six (6H) hidden layers (H stands for hidden layer). We present the evolution of training error in Figure 3. By analyzing this figure, it is clear that shallow networks consisting of one or two hidden layers do not perform, and the PINN learning is bound in learning after few thousand epochs. Even one layer with large number of units, e.g., one hidden layer with 640 units (see the magenta line in the right panel of Figure 3), do not lead to better performance as demonstration that depth is more important than breadth in PINN. Deeper networks with more than three layers lead to lower final training errors and improved learning. However, we find that the final training error saturates for PINNs with more than six hidden layers (results not shown here) for

the two test cases. An important aspect for the deployment of PINN in scientific applications is that the performance of PINNs with four and more hidden layers have comparable performance in the first 500 epochs of the Adam and L-BFGS-B optimizers. Taking in account that the PINN computational cost for PINNs increases with the number layers and realistically only few hundred epochs are necessary for PINN to be competitive with HPC solvers, PINNs with four hidden layers provide the best trade-off in terms of accuracy and computational performance.

For the six hidden layers case, we also check the importance of having a large/small number of units at the beginning/end of the network: we consider the performance of PINN with six hidden layers and 10-20-40-80-160-320 and 320-160-80-40-20-10 units per hidden layer, respectively. We find that to have a large number of units at the beginning of the network and small number of units at the end of the network is detrimental to the PINN performance (a six hidden layer network in this configuration has the same performance of a five hidden layer PINN). Instead, to have a small number of units at the beginning of the network and a large number of units at the end of the network is beneficial to the PINN. **This observation hints that initial hidden layers might responsible for encoding the low-frequencies components (fewer points are needed to represent low-frequency signals) and the following hidden layers are responsible for representing higher-frequency components (several points are needed to represent high-frequency signals).** However, more experiments are needed to confirm this hypothesis.

The most impactful parameter for achieving a low training error is the activation function. This fact is expected as activation functions are nothing else than non-linear interpolation functions (similarly to nodal functions in FEM): some interpolation function might be a better fit to represent the different source terms. For instance, sigmoid functions are a good fit to represent non-differentiable source terms exhibiting discontinuities. On the contrary, a smooth tanh activation function can closely represent smooth functions.

We investigate the impact of different activation functions and show the evolution of the training errors in Figure 4. Together with traditional activation function, we also consider the Locally Adaptive Activation Functions (LAAF): with this technique, a scalable parameter is introduced in each layer separately, and then optimized with a variant of stochastic gradient descent algorithm [12]. The LAAF are provided in the DeepXDE DSL. We investigate LAAF with factor of 5 (LAAF-5) and 10 (LAAF-10) for the tanh, swish and sigmoid cases. The LAAF usage is critical to mitigate the *vanishing-gradient* problem.

The activation function's different impact for the two test cases (smooth and non-smooth source terms) is clear when analyzing the results presented in Figure 4. In the smooth source term case, the best activation function is the locally (to the layer) adaptive tanh activation function with factor 5 (LAAF5 - tanh). In the case of the non-smooth source term, the sigmoid activation function outperforms all the other activation functions. In particular, in this case, the best activation function is the locally (to the layer) adaptive sigmoid activation function with factor 10 (LAAF10 - sigmoid).

As we mentioned in Section 2.1, the data size impacts the training errors. Large data sets increase the PINN accuracy but have larger training errors than the training with small data set because of the error definition (see Equation 6). For this reason, the training error should be compared only for training using the same training data set size. We investigate the impact of three different input data size (1- 1,200 points in the domain and 200 on the boundary, 2- 64×64 points in the domain and 2,000 on the boundary, 3- 128×128 points in the domain and 4,000 on the boundary) with three collocation point distributions (uniform, pseudo-random, and Sobol sequence) for the non-smooth source term. We show the results in Figure 5.

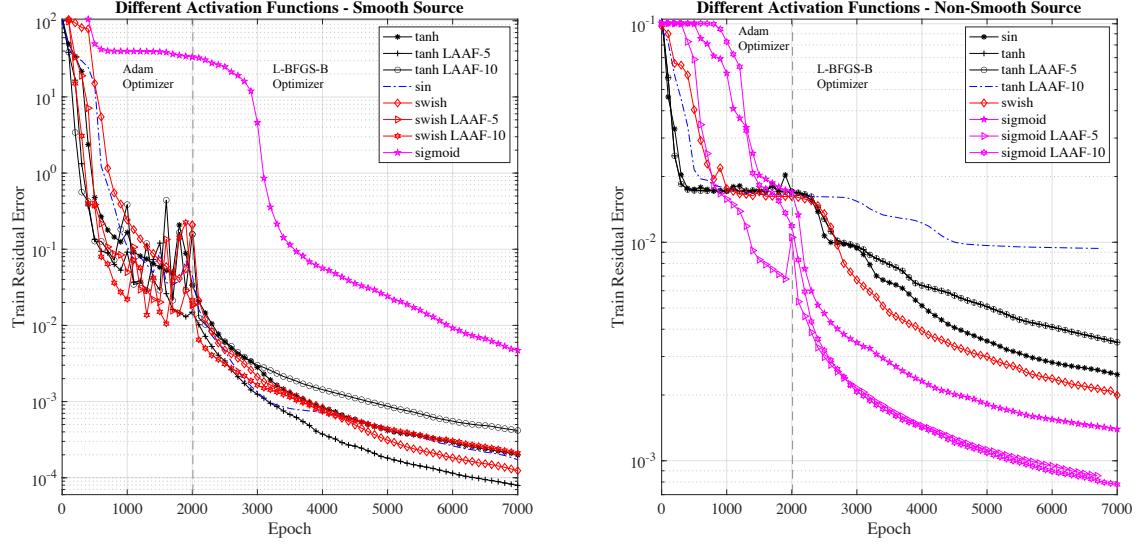


Fig. 4. Training error for different activation functions. The two test cases show rather different performance: the best activation function for smooth source term case is tanh, while it is sigmoid for the non-smooth source term case. Local (to the layer) adaptive activation functions provide a reduction of the training error.

In general, we find that the collocation point distribution does not have a considerable impact on the training error for large data sets: the Sobol and pseudo-random distributions have a slightly better performance than the uniform distribution. For small data sets, pseudo-random distribution result in lower training errors.

We also study the impact of having a *restart* procedure: we train first the PINN with a small data set 1,200 points in the domain and 200 on the boundary) for 4,500 epochs (and then re-train the same network with a large data set (128×128 points in the domain and 4,000 on the boundary) for 2,500 cycles (see the magenta lines and the grey box in Figure 5). Such a restart capability would lead to a large computational saving. However, the results show that to retrain with a large data set does not lead to a decreased error and result in the highest training error.

4 THE IMPORTANCE OF TRANSFER LEARNING

In this study, we found that the usage transfer learning technique is critical for training PINNs with a reduced number of epochs and computational cost. The transfer learning technique consists of training a network solving the Poisson equation with a different source term. We can then initialize the PINN network we intend to solve with the first fully trained network weights and biases. In this way, the first PINN *transfers* the learned information about encoding to the second PINN. To show the advantage of transfer learning in PINN, we solve two additional test cases with smooth and non-smooth source terms. For the test case with the smooth source term, we solve the Poisson equation with source term $f(x, y) = 10(x(x - 1) + y(y - 1)) - 2 \sin(\pi x) \sin(\pi y) + 5(2\pi x) \sin(2\pi y)$.

We initialize the network with the results obtained during the training with Equation 7 as a source term. One of the major advantages of transfer-learning is that we can start the L-BFGS-B optimizer after very few Adam solvers epochs (empirically, we found that 10 Adam epochs ensure that L-BFGS-B will avoid local minima). L-BFGS-B has faster

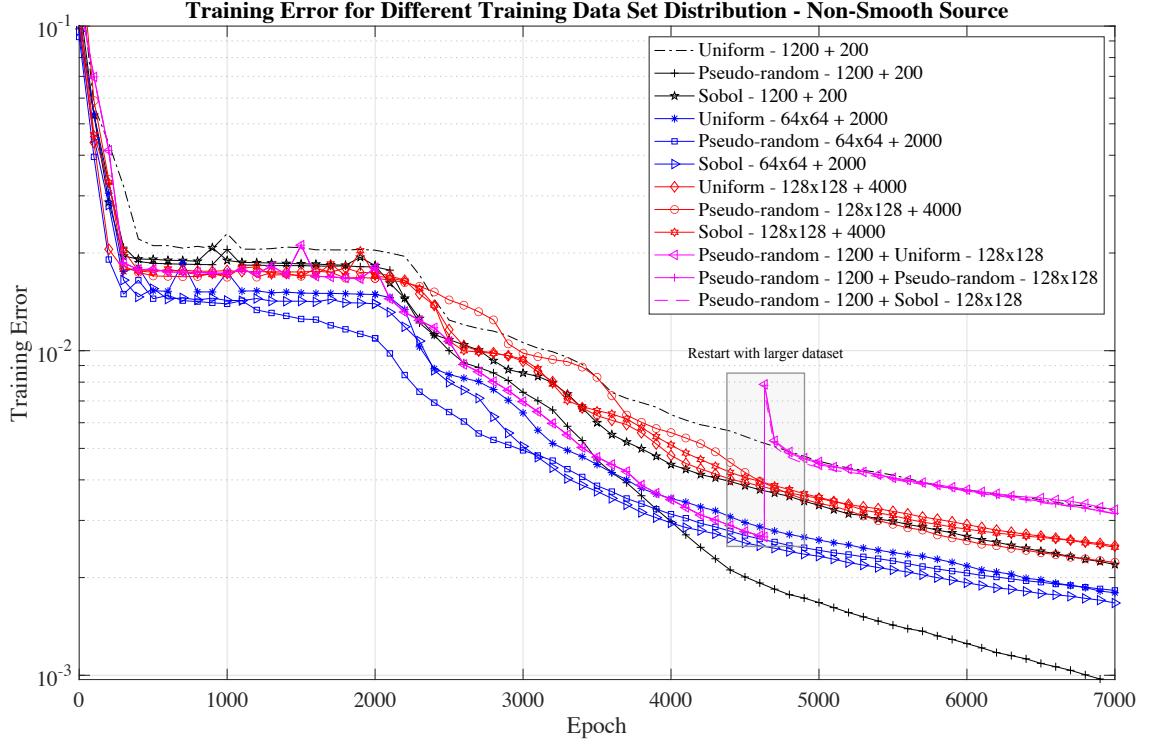


Fig. 5. Training error for different data set (1,200 points in the domain and 200 on the boundary, 64×64 points in the domain and 2,000 on the boundary, 128×128 points in the domain and 4,000 on the boundary) and different distribution (uniform, pseudo-random and Sobol).

convergence than the Adam optimizer and therefore the training is quicker. When not using transfer-learning, we train the PINN with 2,000 epochs of Adam optimizer, followed by 5,000 epochs of L-BFGS-B. When using L-BFGS-B, we perform 10 epochs of Adam optimizer, followed by 6,955 L-BFGS-B epochs.

The black lines in Figure 6 show a comparison of the training error for a network initialized with Xavier weight initialization, e.g., without transfer learning (− black line) and with transfer learning (+ black line). In this case, transfer learning usage allows gaining two orders of improvement in the training error in less than 1,000 epochs.

For the test case with non-smooth source term, we introduce and additional test case solving the Poisson equation with a source term that is everywhere zero except in a circle with radius 0.1 and centered in the x and y coordinates (0.7,0.7).

$$f(x, y) = -10 \text{ for } \sqrt{(x - 0.7)^2 + (y - 0.7)^2} \leq 0.1. \quad (9)$$

For transfer learning, we use the PINN weights obtained training the network to solve the Poisson equation with source term of Equation 9. The blue lines in Figure 6 are the training error without transfer learning. As in the case of smooth-source term, the usage of transfer learning rapidly decreases the training error.

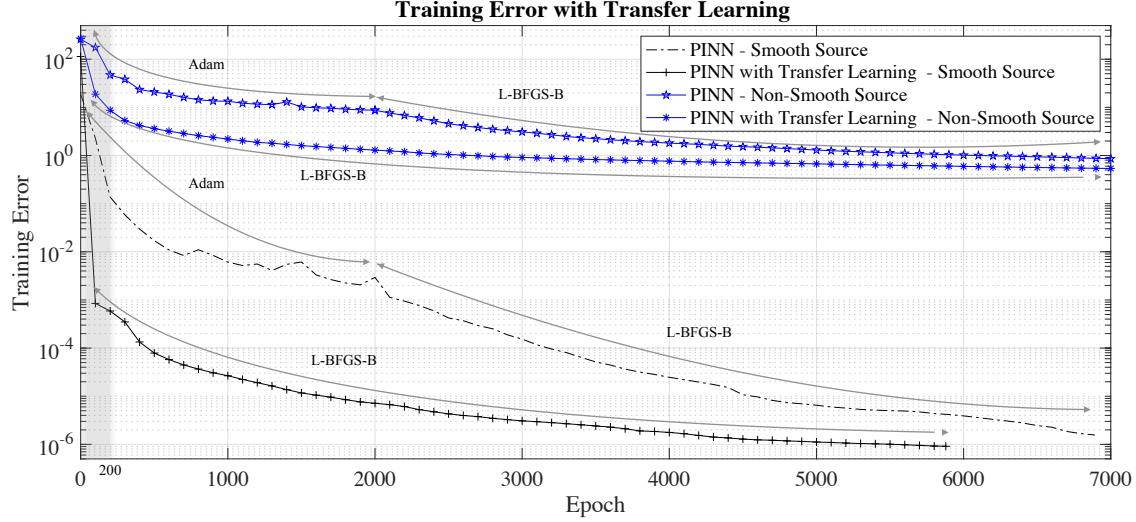


Fig. 6. Training error with and without transfer learning for the smooth and non-smooth source test cases.

We note that usage of the transfer learning leads to an initial (less than 200 L-BFGS-B epochs) *super-convergence* to a relatively low training error. For this reason, **transfer-learning is a necessary operation to make PINN competitive with other solvers used in scientific computing**.

The major challenge for using transfer-learning is to determine which pre-trained PINN to use. In simulation codes, solving the same equation with different source term at each time step, an obvious choice is a PINN that solves the governing equations with a source term at one of the time step. For other cases, we found that PINNs solving problems with source terms containing high-frequency components (possibly more than one component) are suitable for transfer-learning in general situations. We also found that PINNs solving problem with only one low frequency component as source term are not beneficial for transfer learning: their performance is equivalent to the case without transfer learning.

5 THE OLD AND THE NEW: INTEGRATING PINNS INTO TRADITIONAL LINEAR SOLVERS

In Section 2.1, we observed that direct usage of PINN to solve the Poisson equation is still limited by the large number of epochs required to achieve an acceptable precision. One possibility to improve the performance of PINN is to combine PINN with traditional iterative solvers such as the Jacobi, Gauss-Seidel and multigrid solvers [32].

PINN solvers' advantage is the quick convergence to the solution's low frequencies components. However, the convergence to high-frequency features is slow and requires an increasing number of training iteration/epochs. This fact is a result of the the F-principle. Because of this, PINNs are of limited usage when the application requires highly accurate solutions. As suggested by Ref. [47], in such cases, the most viable option is to combine PINN solvers with traditional solvers that can converge rapidly to the solution's high-frequency components (but have low convergence for the low-frequency components). Such methods introduce a computational grid and we compute the differential operators with a finite difference scheme. In this work, we choose the Gauss-Seidel method as it exhibits higher convergence rate than the Jacobi method. Each Gauss-Seidel solver iteration for solving the Poisson equation (Equation

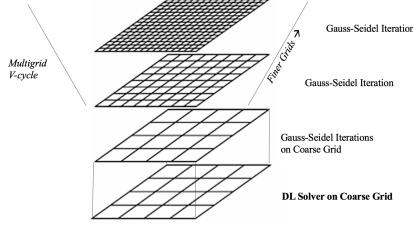


Fig. 7. The hybrid solvers relies on the DL linear solver to determine the solution on a coarse grid that is refined through a multigrid V-cycle performing Gauss-Seidel iterations on finer grids.

2) is:

$$u_{i,j}^{n+1} = 1/4(u_{i+1,j}^n + u_{i-1,j}^n + u_{i,j+1}^n + u_{i,j-1}^n - \Delta x \Delta y f_{i,j}), \quad (10)$$

where i and j are the cell index, Δx and Δy are the grid cell size in the x and y direction, and n is the iteration number. Usually, the Gauss-Seidel method stops iterating when $\|u_{n+1} - u^n\|_2 \leq \delta$, where $\|\dots\|$ is the Euclidean norm and δ is a so-called tolerance and it is chosen as an arbitrarily small value.

Both the Jacobi and Gauss-Seidel methods show fast convergence for small-scale features: this is because the update of unknown values involves only the values of the neighbor points (stencil defined by the discretization of a differential operator). Between two different iterations, the information can only propagate to neighbor cells.

In this work, we combine traditional approaches with new emerging DL methods as shown in Figure 7. Overall, the new solver consists of three phases. We use first the DL PINN solver to calculate the solution on a coarse grid. As second phase, we refine the solution with Gauss-Seidel iterations on the coarse grid until a stopping criteria is satisfied. The third phase is a multigrid V-cycle: we linearly interpolate (or *prolongate* in multigrid terminology) to finer grids and perform a Gauss-Seidel iteration for each finer grid. In fact, several multigrid strategies with different level of sophistications can be sought. However, in this work we focus on a very simple multigrid approach, based on the Gauss-Seidel method and linear interpolation across different grids. The crucial point is that we train a PINN to calculate the solution of the problem on the coarse grid, replacing the multigrid *restriction* (or *injection*) steps in just one phase.

Figure 8 shows a more detailed diagram of a hybrid multigrid solver combining a DL solver to calculate the solution on a coarse grid with a Gauss-Seidel solver to refine the solution and interpolate to finer grid. Because the DL solver converges quickly to the low-frequency coarse-grained components of the solution while high-frequency small-scale components of the solutions are not accurately solved, we perform the training in single-precision floating-point. This would speed-up the training on GPUs (not used in this work) where the number of single-precision floating-point units (FPUs) is higher than CPU.

The hybrid DL solver comprises six basic steps, represented in Figure 8 :

- (1) **Initialize the network weights and biases** - We load from the disk the network structure and initialize the network. To accelerate the convergence, we rely on transfer-learning: we train a network to solve a similar problem and initialize the network. It is important that the same governing equations, boundary conditions and architecture are used. The weights and biases are in single floating-point precision. The time for completing this step is negligible with respect to the total time of the hybrid solver.

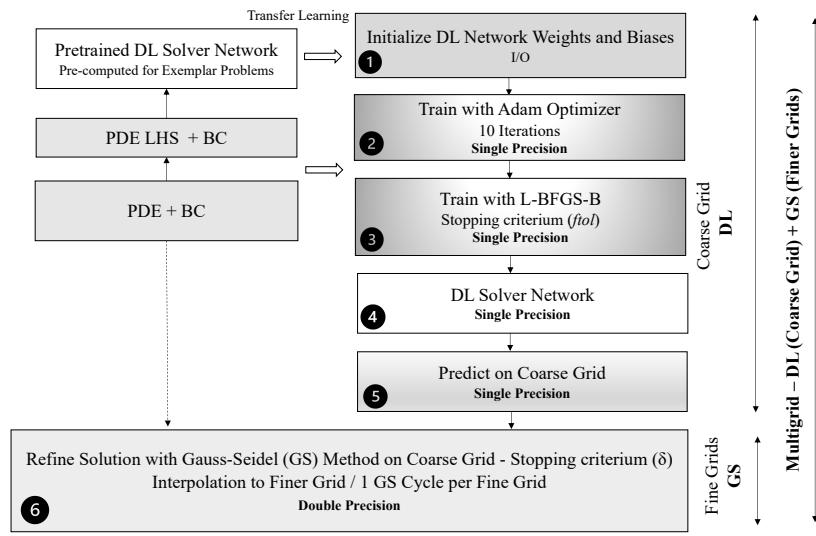


Fig. 8. Structure of the hybrid multigrid solver combining the DL and Gauss-Seidel solvers. Pre-trained networks are pre-computed and used to initialize the DL network. Two main parameters f_{tol}, δ determine the accuracy and the performance of the hybrid solver.

- (2) **Train with Adam Optimizer (10 Epochs)** - We run the Adam optimizer just for a short number of epochs to avoid the consequent L-BFGS-B optimizer converging quickly to the wrong solution (local minimum). By running several tests, we found empirically that only 10 Adams epochs are needed to avoid L-BFGS-B optimizer to converge to the wrong solution. The time for completing this step is typically negligible.
 - (3) **Train with L-BFGS-B Optimizer** - We run the training with the L-BFGS-B optimizer. The stopping criterium is determined by the $ftol$ parameter: the training stops when $(r_k - r_{k+1})/\max(|r_k|, |r_{k+1}|, 1) \leq ftol$, where k is the iteration of the optimizer and r is the value of the function to be optimized (in our case the residual function). Typically, the time for completing the L-BFGS-B dominates is a large part of the execution time of the hybrid solver. To compete with traditional approaches for solving Poisson equation, we set a maximum number of epochs to 1,000.
 - (4) **DL solver is obtained at the end of the training process** - The solver can inference the solution at given collocation points or save it for future transfer-learning tasks, e.g., a simulation repeats the computation of the Poisson equation at different time steps.
 - (5) **The Approximator/Surrogate Network is used to calculate the solution on the coarse grid of the multigrid solver** - We calculate the solution of our problem on the coarse grid of a multigrid solver. This operation is carried with single-precision floating point numbers since high-accuracy is not needed in this step. The result is then cast to double precision for the successive Gauss-Seidel solver. This inference computational time is typically negligible when compared to the total execution time.
 - (6) **Refine the solution with the Gauss-Seidel Method on the coarse grid and interpolate on fine grids** - We perform first Gauss-Seidel iterations to refine the solution on the coarse grid. This solution refinement is critical to remove the vanishing-gradient problem at the boundary. The Gauss-Seidel iteration on the coarse grid stops when $\|u^{n+1} - u^n\|_2 \leq \delta$ where n is the iteration number. After the Gauss-Seidel method stops on the

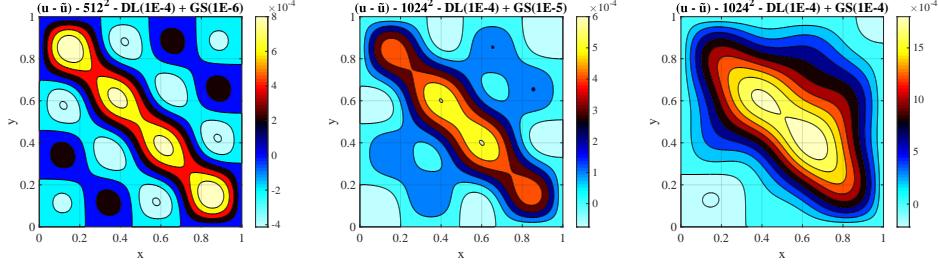


Fig. 9. Hybrid multigrid solver final error ($u - \tilde{u}$) using three different setups: 1 - 512×512 grid with a 64×64 coarse grid, f_{tol} equal to $1E-4$ and δ equal to $1E-6$; 2 and 3 - 1024×1024 grid with a 128×128 coarse grid, f_{tol} equal to $1E-4$ and δ equal to $1E-5$ and $1E-4$.

coarse grid, linear interpolation to finer grids and a Gauss-Seidel iteration per grid are computed. As example, to solve the problem on a 512×512 grid, we perform the following steps: 1) use the DL solver to calculate the solution on 64×64 grid; 2) refine the solution with the Gauss-Seidel method on the 64×64 grid until convergence is reached; 3) carry out a linear interpolation to the 128×128 grid; 4) perform a Gauss-Seidel iteration on the 128×128 grid; 5) carry out a linear interpolation to 256×256 grid; 6) perform a Gauss-Seidel iteration on the 256×256 grid; 7) carry out a linear interpolation to 512×512 grid; 8) perform a final Gauss-Seidel iteration on the 512×512 grid. The interpolation and Gauss-Seidel iterations corresponds to the V-cycle in the multigrid method as shown in Figure 7.

We test the hybrid modified solver against the same problem shown in Section 2.1: we solve the Poisson equation with source term of Equation 7. Leveraging the knowledge gained in the characterization study of Section 3, we use a four hidden layer fully-connected neural network with 50 neurons per hidden layer. To optimize the convergence for solving the Poisson equation with a smooth source term, we rely on LAAF-5 \tanh activation functions: these activations functions provided the best performance in our characterization study. For the transfer learning, we pre-train a network for 2,000 Adam optimizer epochs and 5,000 L-BFGS-B optimizer epochs to solve a Poisson equation with a source term equal to $-2 \sin(\pi x) \sin(\pi y) - 72 \sin(6\pi x) \sin(6\pi y)$. We use an input data set consisting of 100×100 points in the integration domain and 2,000 points on the boundaries for the DL solver. We use the Sobol sequence as training data set distribution. The network weights and biases for transfer learning are saved as checkpoint / restart files in TensorFlow.

For the first test, we employ a 512×512 grid with a 64×64 coarse grid, f_{tol} equal to $1E-4$ and δ equal to $1E-6$. We then test the hybrid multigrid solver on a 1024×1024 grid with a 128×128 coarse grid, f_{tol} equal to $1E-4$ and two values for δ : $1E-5$ and $1E-4$. Figure 9 shows a contour plot the error ($u - \tilde{u}$) for these three configurations. The maximum error for the hybrid multigrid solver is of the $1E-4$ order and less than the error we obtained after an extensive training of a basic PINN (approximately $1E-3$, see the bottom right panel of Figure 2).

Once we showed that the hybrid multigrid solver provides more accurate results than the direct PINN usage, we focus on studying the computational performance. The performance tests are carried out on a 2.9 GHz Dual-Core Intel Core i5, 16 GB 2133 MHz LPDDR3 using macOS Catalina 10.15.7. We use Python 3.7.9, TensorFlow 2.4.0, SciPy 1.5.4 and the DeepXDE DSL. The Gauss-Seidel iteration is implemented in Cython [9] to improve the performance and avoid time-consuming loops in Python. For comparison, we also solve the problem using only the Gauss-Seidel method to solve the problem on the coarse grid and using the petsc4py CG solver. The PETSc version is 3.14.2 and we use r_{tol} (the relative to the initial residual norm convergence tolerance). We repeat the tests five times and report the arithmetic

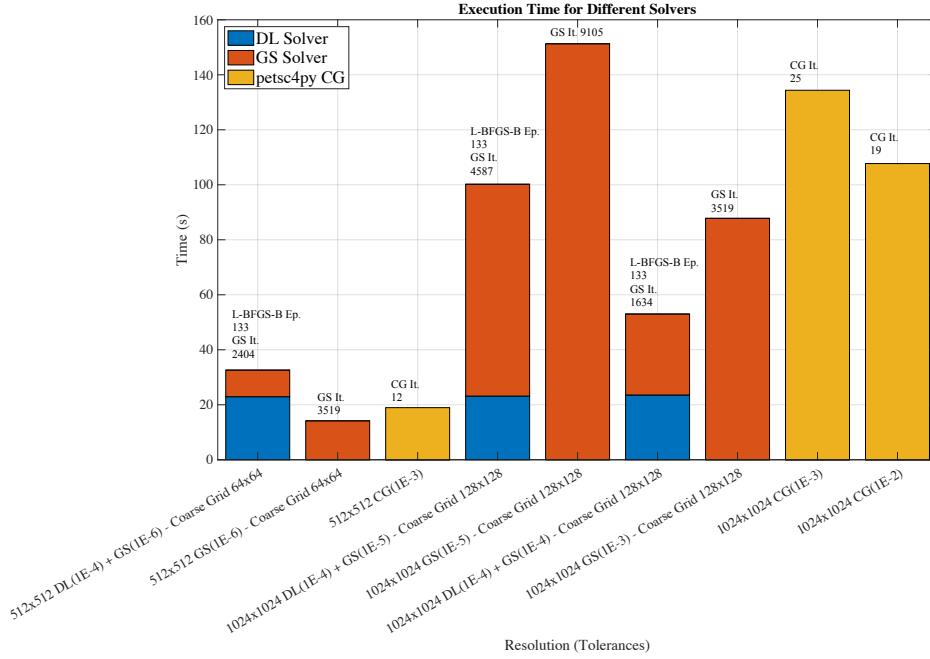


Fig. 10. Execution time, number of epochs and iterations for the hybrid multigrid DL-GS solver and comparison with the performance of a multigrid using only GS and petsc4py CG varying the resolution, and solver stopping criteria. The hybrid multigrid DL-GS solver is faster for problems using larger coarse grids, e.g. 128×128 coarse grids, than the other approaches.

average of the execution times. We do not report error bars as the standard deviation is less than 5% of the average value. Figure 10 shows the execution time together with number of epochs and iterations for the three different configurations.

The most important result is that by using an optimized configuration, transfer learning, and integrating DL technologies into traditional approaches, we can now solve the Poisson equation with an acceptable precision with a reduced number of training iterations. This reduction of number of training epochs translates to complete the problem, presented in Section 2.1, in less than few minutes instead of hours (see Figure 2) on the Intel i5 system. While the execution depends on the specific hardware platform and implementation, the number of training epochs and GS iterations on the coarse grid (reported on the top of the histogram bars in Figure 10) are not. Overall, we found that 133 epochs are needed for the L-BFGS-B optimizer to reach an *ftol* equal to 1E-4.

Figure 10 histograms also show the breakdown between the time spent in the DL and Gauss-Seidel solvers used in the multigrid V-cycle. Note that the execution time for the DL solver is approximately the same for calculating the values on the two coarse grids: 64×64 and 128×128. This is because of PINN are *gridless* methods: only the negligible inference computational cost is different. For comparison, we show the performance of the Gauss-Seidel solver for the coarse grid (orange bars) and py4petsc CG solver petsc4py (yellow bars) with different *rtol* values. When the coarse grid is small, e.g., 64×64, the cost for training the DL solver is higher than using a basic method such Gauss-Seidel: using the Gauss-Seidel method for the coarse grid is faster than using the DL solver for the coarse grid. However, for

larger coarser grids, e.g., 128×128 , the hybrid multigrid solver is fastest. For comparison, we present the results obtained running the `petsc4py` CG with different *rtol* values. Overall, the performance of the hybrid solver is competitive with state-of-the-art linear solvers. We note that none of the methods and codes have been optimized nor compared at same accuracy (the stopping criteria are defined differently for different solvers), so the performance results provide an indication of potential of the hybrid solver without providing absolute performance values.

6 DISCUSSION AND CONCLUSION

This paper presented a study to evaluate the potential of emerging new DL technologies to replace or accelerate old traditional approaches when solving the Poisson equation. We show that directly replacing traditional methods with PINNs results in limited accuracy and a long training period. Setting up an appropriate configuration of depth, activation functions, input data set distribution, and leveraging transfer-learning could effectively optimize the PINNs solver. However, PINNs are still far from competing with HPC solvers, such as PETSc CG. In summary, PINNs in the current state cannot yet replace traditional approaches.

However, while the direct usage of PINN in scientific applications is still far from meeting computational performance and accuracy requirements, hybrid strategies integrating PINNs with traditional approaches, such as multigrid and Gauss-Seidel methods, are the most promising option for developing a new class of solvers in scientific applications. We showed the first performance results of such hybrid approaches on the par (and better for large coarse grids) with other state-of-the-art solver implementations, such as PETSc.

When considering the potential for PINNs of using new emerging heterogeneous hardware, PINNs could benefit from the usage of GPUs that are workforce for DL workloads. It is likely that with the usage of GPUs, the performance of hybrid solvers can outperform state-of-the-art HPC solvers. However, PINN DSL frameworks currently rely on SciPy CPU implementation of the key PINN optimizer, L-BFGS-B, and its GPU implementation is not available in SciPy. The new TensorFlow 2 Probability framework¹ provides a BFGS optimizer that can be used on GPUs. Another interesting research direction is investigating the role and impact of the low and mixed-precision calculations to train the approximator network. The usage of low-precision formats would allow us to use tensorial computational units, such as tensor cores in Nvidia GPUs [21] and Google TPUs [13], boosting the DL training performance.

From the algorithmic point of view, a line of research we would like to pursue is a better and more elegant integration of the DL into traditional solvers. One possibility is to extend the seminal work on discrete PINNs [35] combining Runge-Kutta solvers and PINN for ODE solutions: a similar approach could be sought to encode information about discretization points into PINN. However, currently, this approach is supervised and requires the availability of simulation data. In addition, the development of specific network architectures for solving specific PDEs is a promising area of research. A limitation of this work is that we considered only fully-connected networks as surrogate network architectures. For solving the Poisson equation and elliptic problems in general, the usage of convolutional networks with large and dilated kernels is likely to provide better performance of fully-connected DL networks to learn non-local relationships a signature of elliptic problems [20].

The major challenge is integrating these new classes of hybrid DL and traditional approaches, developed in Python, into large scientific codes and libraries, often written in Fortran and C/C++. One possibility is to bypass the Python interface of major DL frameworks and use their C++ runtime directly. However, this task is complex. An easier path for the software integration of DL solvers into legacy HPC applications is highly needed.

¹<https://www.tensorflow.org/probability>

Despite all these challenges and difficulties ahead, this paper shows that the integration of new PINNs DL approaches into *old* traditional HPC approaches for scientific applications will play an essential role in the development of next-generation solvers for linear systems arising from differential equations.

ACKNOWLEDGMENTS

Funding for the work is received from the European Commission H2020 program, Grant Agreement No. 801039 (EPiGRAM-HS).

REFERENCES

- [1] ABADI, M., BARHAM, P., CHEN, J., CHEN, Z., DAVIS, A., DEAN, J., DEVIN, M., GHEMAWAT, S., IRVING, G., ISARD, M., ET AL. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)* (2016), pp. 265–283.
- [2] BALAY, S., ABHYANKAR, S., ADAMS, M., BROWN, J., BRUNE, P., BUSCHELMAN, K., DALCIN, L., DENNER, A., EIJKHOUT, V., GROPP, W., ET AL. Petsc users manual.
- [3] BAYDIN, A. G., PEARLMUTTER, B. A., RADUL, A. A., AND SISKIND, J. M. Automatic differentiation in machine learning: a survey. *Journal of machine learning research* 18 (2018).
- [4] DALCIN, L. D., PAZ, R. R., KLER, P. A., AND COSIMO, A. Parallel distributed computing using python. *Advances in Water Resources* 34, 9 (2011), 1124–1139.
- [5] ESMAEILZADEH, S., AZIZZADENESHELI, K., KASHINATH, K., MUSTAFA, M., TCHELEPI, H. A., MARCUS, P., PRABHAT, M., ANANDKUMAR, A., ET AL. Meshfreeflownet: A physics-constrained deep continuous space-time super-resolution framework. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis* (2020), IEEE, pp. 1–15.
- [6] FLETCHER, R. *Practical methods of optimization*. John Wiley & Sons, 2013.
- [7] GAO, H., SUN, L., AND WANG, J.-X. Phygeonet: Physics-informed geometry-adaptive convolutional neural networks for solving parametric pdes on irregular domain. *arXiv preprint arXiv:2004.13145* (2020).
- [8] GOODFELLOW, I., BENGIO, Y., COURVILLE, A., AND BENGIO, Y. *Deep learning*, vol. 1. MIT press Cambridge, 2016.
- [9] GORELICK, M., AND OZSVALD, I. *High Performance Python: Practical Performant Programming for Humans*. O'Reilly Media, 2020.
- [10] GULLI, A., AND PAL, S. *Deep learning with Keras*. Packt Publishing Ltd, 2017.
- [11] HAGHIGHAT, E., AND JUANES, R. Sciamn: A keras wrapper for scientific computations and physics-informed deep learning using artificial neural networks. *arXiv preprint arXiv:2005.08803* (2020).
- [12] JAGTAP, A. D., KAWAGUCHI, K., AND EM KARNIADAKIS, G. Locally adaptive activation functions with slope recovery for deep and physics-informed neural networks. *Proceedings of the Royal Society A* 476, 2239 (2020), 20200334.
- [13] JOUPPI, N. P., YOUNG, C., PATIL, N., PATTERSON, D., AGRAWAL, G., BAJWA, R., BATES, S., BHATIA, S., BODEN, N., BORCHERS, A., ET AL. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture* (2017), pp. 1–12.
- [14] KELLEY, C. T. *Iterative methods for linear and nonlinear equations*. SIAM, 1995.
- [15] KHARAZMI, E., ZHANG, Z., AND KARNIADAKIS, G. E. Variational physics-informed neural networks for solving partial differential equations. *arXiv preprint arXiv:1912.00873* (2019).
- [16] KHARAZMI, E., ZHANG, Z., AND KARNIADAKIS, G. E. hp-vpinns: Variational physics-informed neural networks with domain decomposition. *arXiv preprint arXiv:2003.05385* (2020).
- [17] KUMAR, S. K. On weight initialization in deep neural networks. *arXiv preprint arXiv:1704.08863* (2017).
- [18] LAGRASIS, I. E., LIKAS, A., AND FOTIADIS, D. I. Artificial neural networks for solving ordinary and partial differential equations. *IEEE transactions on neural networks* 9, 5 (1998), 987–1000.
- [19] LU, L., MENG, X., MAO, Z., AND KARNIADAKIS, G. E. Deepxde: A deep learning library for solving differential equations. *arXiv preprint arXiv:1907.04502* (2019).
- [20] LUNA, K., AND BLASCHKE, J. Accelerating gmres with deep learning in real-time. *Supercomputing Posters* (2020).
- [21] MARKIDIS, S., DER CHIEN, S. W., LAURE, E., PENG, I. B., AND VETTER, J. S. Nvidia tensor core programmability, performance & precision. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)* (2018), IEEE, pp. 522–531.
- [22] MARKIDIS, S., LAPENTA, G., AND RIZWAN-UDIN. Multi-scale simulations of plasma with ipic3d. *Mathematics and Computers in Simulation* 80, 7 (2010), 1509–1519.
- [23] MISHRA, S., AND MOLINARO, R. Estimates on the generalization error of Physics Informed Neural Networks (PINNs) for approximating PDEs. *arXiv e-prints* (June 2020), arXiv:2006.16144.
- [24] MISHRA, S., AND MOLINARO, R. Estimates on the generalization error of physics informed neural networks (PINNs) for approximating PDEs ii: A class of inverse problems. *arXiv preprint arXiv:2007.01138* (2020).
- [25] MORTON, K. W., AND MAYER, D. F. *Numerical solution of partial differential equations: an introduction*. Cambridge university press, 2005.
- [26] NASCIMENTO, R. G., AND VIANA, F. A. Fleet prognosis with physics-informed recurrent neural networks. *arXiv preprint arXiv:1901.05512* (2019).

- [27] OFFERMANS, N., MARIN, O., SCHANEN, M., GONG, J., FISCHER, P., SCHLATTER, P., OBABKO, A., PEPLINSKI, A., HUTCHINSON, M., AND MERZARI, E. On the strong scaling of the spectral element solver nek5000 on petascale systems. In *Proceedings of the Exascale Applications and Software Conference 2016* (2016), pp. 1–10.
- [28] PANG, G., LU, L., AND KARNIADAKIS, G. E. fpinn: Fractional physics-informed neural networks. *SIAM Journal on Scientific Computing* 41, 4 (2019), A2603–A2626.
- [29] PASZKE, A., GROSS, S., MASSA, F., LERER, A., BRADBURY, J., CHANAN, G., KILLEEN, T., LIN, Z., GIMELSHEIN, N., ANTIGA, L., ET AL. Pytorch: An imperative style, high-performance deep learning library. *arXiv preprint arXiv:1912.01703* (2019).
- [30] PAUL F. FISCHER, J. W. L., AND KERKEMEIER, S. G. nek5000 Web page, 2008. <http://nek5000.mcs.anl.gov>.
- [31] PSICHOGIOS, D. C., AND UNGAR, L. H. A hybrid neural network-first principles approach to process modeling. *AIChE Journal* 38, 10 (1992), 1499–1511.
- [32] QUARTERONI, A., SACCO, R., AND SALERI, F. *Numerical mathematics*, vol. 37. Springer Science & Business Media, 2010.
- [33] RAISSI, M., PERDIKARIS, P., AND KARNIADAKIS, G. E. Physics informed deep learning (part i): Data-driven solutions of nonlinear partial differential equations. *arXiv preprint arXiv:1711.10561* (2017).
- [34] RAISSI, M., PERDIKARIS, P., AND KARNIADAKIS, G. E. Physics informed deep learning (part ii): Data-driven discovery of nonlinear partial differential equations. *arXiv preprint arXiv:1711.10566* (2017).
- [35] RAISSI, M., PERDIKARIS, P., AND KARNIADAKIS, G. E. Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *Journal of Computational Physics* 378 (2019), 686–707.
- [36] RAISSI, M., YAZDANI, A., AND KARNIADAKIS, G. E. Hidden fluid mechanics: Learning velocity and pressure fields from flow visualizations. *Science* 367, 6481 (2020), 1026–1030.
- [37] RAMACHANDRAN, P., ZOPH, B., AND LE, Q. V. Searching for activation functions. *arXiv preprint arXiv:1710.05941* (2017).
- [38] RAO, S. S. *The finite element method in engineering*. Butterworth-heinemann, 2017.
- [39] RUBINSTEIN, R. Y., AND KROESE, D. P. *Simulation and the Monte Carlo method*, vol. 10. John Wiley & Sons, 2016.
- [40] SOBOŁ, I. Quasi-monte carlo methods. *Progress in Nuclear Energy* 24, 1-3 (1990), 55–61.
- [41] TROTTERBERG, U., OOSTERLEE, C. W., AND SCHULLER, A. *Multigrid*. Elsevier, 2000.
- [42] VAN DER SPOEL, D., LINDAHL, E., HESS, B., GROENHOF, G., MARK, A. E., AND BERENDSEN, H. J. Gromacs: fast, flexible, and free. *Journal of computational chemistry* 26, 16 (2005), 1701–1718.
- [43] VIRTANEN, P., GOMMERS, R., OLIPHANT, T. E., HABERLAND, M., REDDY, T., COURNAPEAU, D., BUROVSKI, E., PETERSON, P., WECKESSER, W., BRIGHT, J., ET AL. Scipy 1.0: fundamental algorithms for scientific computing in python. *Nature methods* 17, 3 (2020), 261–272.
- [44] WANG, C., BENTIVEGNA, E., ZHOU, W., KLEIN, L., AND ELMEGREEN, B. Physics-informed neural network super resolution for advection-diffusion models. *arXiv preprint arXiv:2011.02519* (2020).
- [45] WANG, S., TENG, Y., AND PERDIKARIS, P. Understanding and mitigating gradient pathologies in physics-informed neural networks. *arXiv preprint arXiv:2001.04536* (2020).
- [46] WEISS, K., KHOSHGOFTAAR, T. M., AND WANG, D. A survey of transfer learning. *Journal of Big data* 3, 1 (2016), 1–40.
- [47] XU, Z.-Q. J., ZHANG, Y., LUO, T., XIAO, Y., AND MA, Z. Frequency principle: Fourier analysis sheds light on deep neural networks. *arXiv preprint arXiv:1901.06523* (2019).
- [48] ZHU, K., AND MÜLLER, E. A. Generating a machine-learned equation of state for fluid properties. *The Journal of Physical Chemistry B* 124, 39 (2020), 8628–8639.